Random Number Generators:

Introduction for Operating System Developers

Key material generation is as important to strong cryptosystems as the algorithms used. Weak random number generators (RNGs) have been known to create key material that is guessable by adversaries[1], making the strength of the algorithms irrelevant in cryptographic attacks. This paper, intended for operating system developers, provides an overview of considerations developers should be making when designing and using RNGs, outlines how RNGs work, and gives recommendations for developing and using RNGs.

Why Good RNGs Are Important

Consider an example system using 256-bit AES to protect sensitive communications. Imagine these keys are generated by selecting from a hard-coded list of four possible values. If an attacker knows this list, then he needs at most four guesses to read the sensitive traffic, instead of the 2^256 guesses that should be required to break the AES algorithm. This situation is equivalent to using a weak RNG because an attacker can exhaust the key space.

When designing a security product, there are many situations in which "random" numbers are needed. We can divide these situations into two classes – key material and one-time numbers:

Key Material. Key material includes authentication keys (RSA, DSA, ECDSA), key agreement parameters (DH and ECDH), and encryption keys (3DES, AES, RSA). As the example above indicates, the reason key material needs a good RNG is to prevent guessing. So while an incrementing counter may avoid repeats, an adversary can guess the next output and, thus, guess the key material.

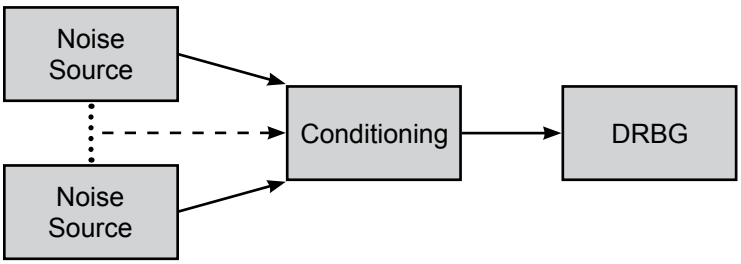
One-Time Numbers. One-time numbers include unique identifiers, session IDs, nonces, and most initialization vectors. With this class, the goal of using an RNG is not to prevent guessing, but to avoid repeats. In these situations, a predictable output, while not recommended, may be acceptable.

What Makes Good RNG Output

Good RNGs produce bytes that are unpredictable. The measurement of this unpredictability is called "entropy." Good RNGs are guaranteed to produce outputs with as much entropy as requested - that is, in a sequence of bits, each bit is equally likely to be zero as one. System services should request as much entropy as the security strength[2] of the algorithm.

How a Good RNG Works

A good RNG usually has three pieces – (1) noise source(s), (2) optional conditioning block, and (3) deterministic random bit generator (DRBG). The RNG should be designed and provided by the hardware or operating system rather than being implemented by application developers.



Noise Sources. The noise source is what ultimately ensures that your RNG output is not guessable. The entropy in an RNG should be measured by testing the noise sources. NIST has published entropy tests[3] to estimate the entropy in noise sources. Additionally, some real-time testing should be performed on the source to guarantee that it is operating correctly.

Many noise sources produce less than one bit of entropy per output; that's ok – the RNG must compensate by gathering a lot of output from this source, possibly requiring a long time. For this reason, and because noise sources may fail, having multiple noise sources is recommended.

[1] Durumeric, Z., Halderman, J., Heninger, N., Wustrow, E. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices." In *Proc. 21st USENIX Security Symposium*, Aug 2012. Rev. 2.

[2] Security Strength is not necessarily the size of the key. NIST SP800-57 describes security strength and gives the strengths for various algorithms.

[3] NIST SP800-90B Draft Standard

Conditioning. The output from the noise source then goes through "mixing and whitening." "Mixing" is simply the addition of data from the noise source into the pool to be used by the DRBG. "Whitening" is a process by which the noise source output or pool is modified so that each bit has equal probability of being zero or one. Usually this is accomplished with a hash function. Mixing and whitening can also be accomplished by the DRBG step, so a separate conditioning step is not always necessary.

Entropy tests are useless after this step because the whitening is a one-way function. At this point, the data is sufficient for use for key material and could be the output of an RNG; however, the next call to the RNG would require gathering more output from the noise sources – likely making such an RNG very slow. Use of a DRBG is therefore recommended to improve the speed of the RNG.

DRBG. Deterministic Random Bit Generators (also known as Pseudo Random Number Generators – PRNGs) take input (a seed) from either the noise source(s) or the conditioning step and produce outputs of random values.

Good DRBGs are designed not to leak information about the secret "state" of the RNG for a large but limited number of iterations. The state is the information required to mathematically compute the next output of the RNG. Bad DRBGs, such as Linear Congruential Generators used in most C rand() and random() implementations, leak state and are not suitable to cryptographic applications.[4] NIST SP800-90A gives algorithmic descriptions of several good DRBGs, their security strengths, and the necessary input sizes.[5]

When the DRBG has output the maximum number of bytes it can securely output, the DRBG must be "reseeded." Here, reseed really means "augment" - mixing more entropy from noise sources into the DRBG state.

The best implementations reseed often (for example, at every call to the RNG) – long before hitting the output limit.

DRBGs can be chained together to produce key material, but the security strength of the final output is only as cryptographically strong as that of the weakest link.

Because generating good random can be an expensive operation both in time and calculation, developers can make trade-offs by using a DRBG, thereby reserving the good RNG for the key material.

Recommendations

- OS RNG implementations should be Common Criteria and/or FIPS 140-2 validated.
- An RNG API should be available to all applications for requesting bytes from the RNG with a requested security strength.
- The RNG API should allow applications to contribute bytes to the entropy pool. These bytes do not count as entropy.
- Adding hardcoded device-unique values can help prevent RNG repeats across product lines, but these values do not count as entropy.
- Similarly, saving the state of the entropy pool between shutdown and startup can prevent RNG repeats on a single device, but this state does not count as unpredictable entropy.
- Mobile Device OSes have lots of additional software noise sources available; in particular mobile devices have baseband, accelerometer, camera, and microphone activity which may be used as sources.
- Network Device OSes have very few software noise sources available and, as such, should use a hardware noise source(s).

[4] Reeds, J. "Cracking a Random Number Generator" In *Cryptologia*, Jan. 1977. Vol. 1 No. 1; p. 20-26. And Matsumoto, M. and Nishimura, T. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." In *ACM Transactions on Modeling and Computer Simulation*, Jan. 1998. Vol. 8, No. 1; p. 3-30.

[5] NIST SP800-90A